Singularity Design Note 37

Singularity ASMP

In which we describe Singularity ASMP design and the current status of ASMP support.

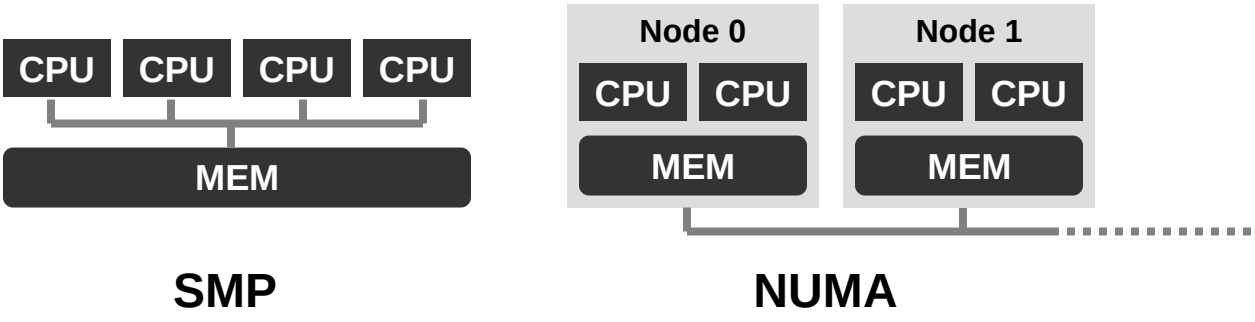
1. Introduction

This design note describes the current Singularity ASMP design and the status of the ASMP supports added to Singularity.

2. Motivation

2.1. SMP and NUMA

Currently there are two types of multiprocessor systems in commodity hardware. The first one is Symmetric Multiprocessor System (SMP) in which the processors share common I/O and memory buses. Technically, they all have equal access to resources. However, memory contention could be the major bottleneck, especially when we have more and more processors.



SMP NUMA

The second trend is NUMA architecture (non-uniform memory architecture systems) which holds the high end of the multiprocessor market. In NUMA architecture, processors are arranged into nodes that have their own local memory. For example, the figure above depicts the nodes arrangement in which each node has two processors, and each node has its own local memory. Furthermore, a processor is still allowed to access memory in other nodes, however for performance reason, it is very recommended that such system access local memory from the same node rather than remote memory from other nodes.

2.2. Heterogeneous MP

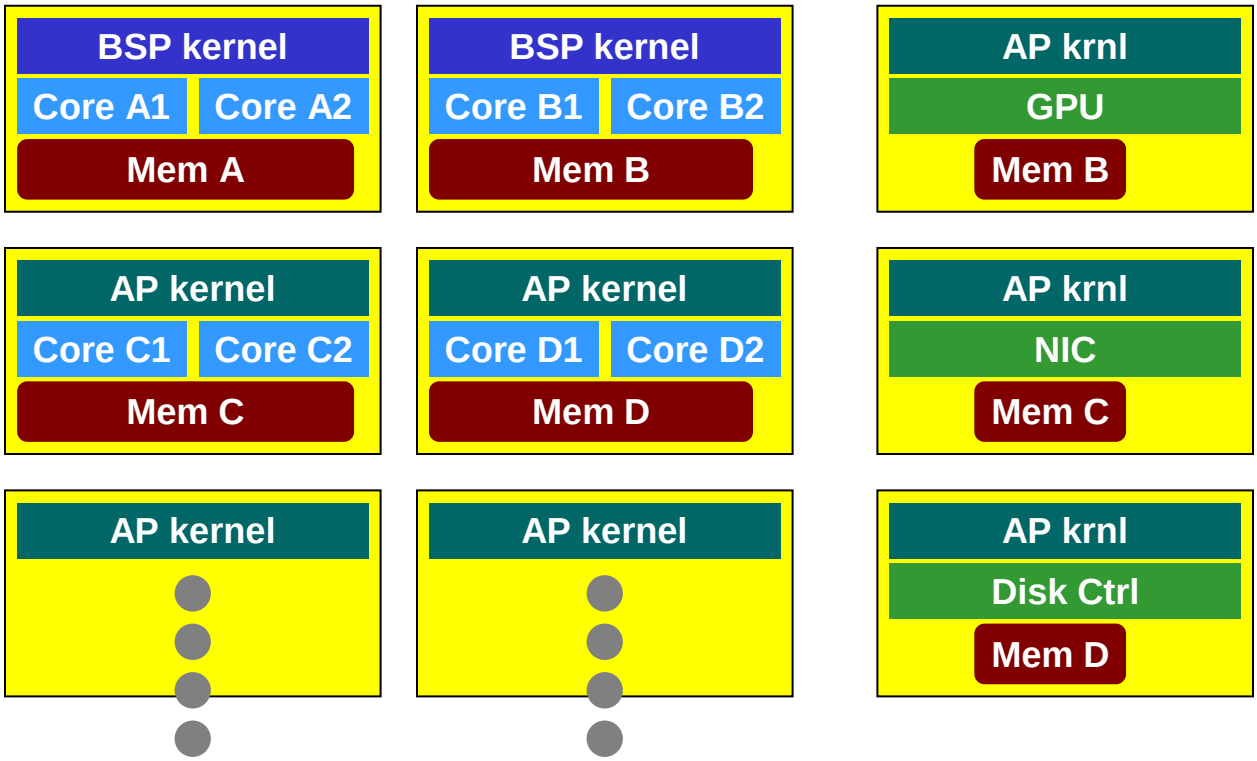
Beyond NUMA architecture, current trend also shows the prevalence of heterogeneous multiprocessor as seen in modern desktop and servers. Even in desktop, besides having the general purpose CPU, we could have graphic processor (GPU), network processors (NP) on NICs, RAID controllers, etc., which used to be special purpose processors, but now there is a trend towards replacing some of those with embedded general purpose processors.

The best way to leverage this model is to assign specific application type to specific processors. We could run the disk driver process on the controller, or even the file system instances on the controller. For a network controller, we could run packet filtering on the card to early deciding which processor needs the packet [McAuley'03, Muir'98].

Besides heterogeneity in terms of functionality, we also have heterogeneity in terms of processor's capability. For example, a processor could have smaller cache size, or smaller superscalar width in order to reduce power dissipation. This type work has been shown by some researches from UCSD and HP Labs [Kumar'03]. One of the motivations for their proposal is that different applications have different resource requirements during their execution, hence the need of heterogeneous processor in terms of capability.

3. Singularity ASMP Design

Due to the motivation describe above we have picked an ASMP design that is neither SMP nor NUMA. The figure below illustrates the ASMP design that will be described thoroughly in this section.



The left part of the figure basically shows the NUMA nodes (internal processors), while the right part of the figure shows the external processors such as the GPU, NC, and Disk Controller. Each NUMA node in

this case has two processors. Note that internal processors are not uniform as suggested in the Heterogeneous MP section above.

Two main characteristics of Singularity ASMP design are:

1. **BSP and AP:** We distinct two kinds of processors that we associate to existing processors: the Boot Strap Processor and the Application Processor. On the BSP, we run a full kernel. On the AP, we run a proxy kernel, which is a cut-down version of the full kernel.
2. **Exclusive memory per kernel:** We partition the memory so that each kernel has its exclusive address space with an exception that the BSP kernel can access all memory without restriction.

3.1. Boot Strap Processor and BSP Kernel

BSP: The Boot Strap Processor is expected to be assigned to processor(s) with the highest and most complete capabilities among the other processors. External processors such as GPU cannot be assigned as BSP. A full version of the kernel runs on the BSP.

BSP kernel: A BSP kernel can be given one or more BSP processors, preferably located in the same node. This will allow concurrent threads working on the same piece of memory. When a machine has tens or hundreds of processors, more than one BSP kernels might exist in the system. However, each BSP kernel has its own address space which could improve fault isolation. The BSP kernel also has the ability to manage all memory and has offloading capability where it sends jobs to the AP kernels which will run the jobs on APs.

BSP Scheduler: In Singularity ASMP design, we treat the processors and kernels like a distributed system. The rule of the thumb in distributed system is a very loosely couple design. From that perspective, the schedulers are designed to be loosely coupled. In other words, the schedulers in the BSP kernels are different than the schedulers in AP kernels, and they are pretty much loosely coupled.

The BSP schedulers act like a full-fledge scheduler. They manage at process-level and have knowledge of all threads but do not have thread states. They minimally know when those threads start and exit. With this design, when we execute “ps” for example, the BSP does not need to send tens or hundreds of inter-processor interrupts (IPI) to the AP kernels. The AP scheduler is explained in next section.

A further discussion is whether BSP scheduler should interfere AP scheduler and in what circumstances it is allowed to interfere. In other words, we could treat BSP scheduler as the big brother, and whenever something is wrong in the system, e.g. AP is overloaded, the BSP could interfere and load balance the processes again.

Note: Take a look at gang scheduling.

3.2. Application Processor and AP Kernel

AP: There are two types of Application Processors: internal and external processors (e.g. GPU). The key difference between the two types of processors is how the memory is assigned to them. For external processors, some memory chunks from NUMA nodes’ memory are assigned to them. Each external processor has its own exclusive address space that is not shared with other APs. For internal AP processors, one or more NUMA processors are grouped to form an AP kernel. Each group has its own exclusive address space that is not shared with other groups. The reason for the latter design is we want to allow concurrent threads working on the same piece of memory.

AP Kernel: An AP kernel runs on top of an external processor or a group of internal APs. Since AP kernel only runs application and does not manage the whole system, then we need to decide the minimal capabilities that an AP kernel should have. For example, it could only have a small local scheduler that

does not know the existence of other threads running on other processors. It could only manage its own memory.

AP as Proxy ABI: Since AP kernel is not a full kernel, it cannot serve all ABI calls made by applications running on AP. Hence, another responsibility for AP kernel is to reroute some ABI calls to the BSP kernel. As AP kernel manages its own memory, some ABI calls related to memory management will be handled locally.

AP Scheduler: The AP scheduler is the cut-down version of the full-fledge scheduler. It only manages at thread-level and has the ability to switch tasks between local threads. However, it does not have knowledge of non-local threads due to the loosely-coupled reason. The scheduler in AP will have synchronization primitives, wait handles, etc.

Memory: In terms of the interaction between AP kernels, we are not planning to have any shared memory at all as we want to reduce contention in anticipating the growth of number of processors. Some works have suggested "loan concept" which allow a processor to borrow page from another processors. They do that mainly to have a competitive performance [Chapin'95].

3.3. Advantages and Disadvantages

The two main advantages in Singularity ASMP design are fault isolation and performance isolation. In terms of unit of failure, the unit of failure for SMP system is the whole system because any significant hardware and software fault causes the entire system to crash as a processor could potentially corrupt all memory. This unit of failure is obviously unacceptable for large-scale machines. On a strict NUMA OS where a node cannot access remote memory in another node, the unit of failure is per-node; a processor fault will not make other computations on another node fail. Nevertheless, if a node has a lot of processors, the unit of failure is still too big.

In Singularity ASMP design, the unit of failure is the number of processors assigned to a kernel. A kernel crash will not make other kernel crash with the exception of BSP kernel crash; Since BSP kernel can manage all memory in the system, it could potentially corrupt the whole system. However, an AP kernel crash will not cause other AP kernels fail. When we have huge number of processors we might want to make groups of AP kernels, where each group is managed by one BSP kernel. Hence in the case of a BSP kernel crash, the whole system will not be dead; instead other BSPs could take control temporarily. This approach is well known in storage processors where a storage processor manage a certain number of disks and when it fails another storage processor will temporarily take control of the disks belonging to the failed processor.

The second advantage is performance isolation. In particular, the performance of application executed by an AP kernel is not directly impacted by the processing performed by other AP kernels. A good illustration of this problem is an analysis done by Balakrishnan et. al. [Balakrishnan'05]. They find that if we have a mix of fast and slow processors having threads that share memory, then the processing performance is reduced to that of the slowest processors.

One potential disadvantage is memory shortage. If memory does not scale up to the number of processors, we could have shortage of memory as we divide the memory across the BSP and AP kernels. Furthermore, today's application working set is quite large, hence leading to trashing.

4. Singularity ASMP Support

This section describes the current status of ASMP support in Singularity. In particular, there are three supports that have been added: memory partitioning, offloading task to AP, and ABI shim.

The first one is to simply extract the processor, cache, and memory hierarchy information so that it is accessible from the kernel and a diagnostic application. This information will be used by the boot process to partition the memory across processors.

The second support is having the ability run applications on AP processors. This involves adding kernel support that will prepare the corresponding application process in different part of the memory for the other processor to run.

The third support, ABI shim, allows applications running on AP processors to make ABI calls.

The other supports that are need to be done are: AP scheduler that switches tasks between threads for use by application processors and MP channel architecture that needs to be extended to allow safe execution for multi-processors system.

4.1. Partitioning Memory

The boot process parses NUMA information from the ACPI SRAT (System Resource Affinity Table) table that contains topology information for all the processors and memory present in the system. Currently, the boot process partitions the memory such that a processor in one domain could not get memory from another domain. After examining some big NUMA SRAT tables from the production team, this decision is conflicting. From the big SRAT tables, some domains might have disabled processors but still have enabled memory. This memory should be used by other domains. In other words, domain number should not be implied as restriction. Instead, it might be better to see it as hints for better performance.

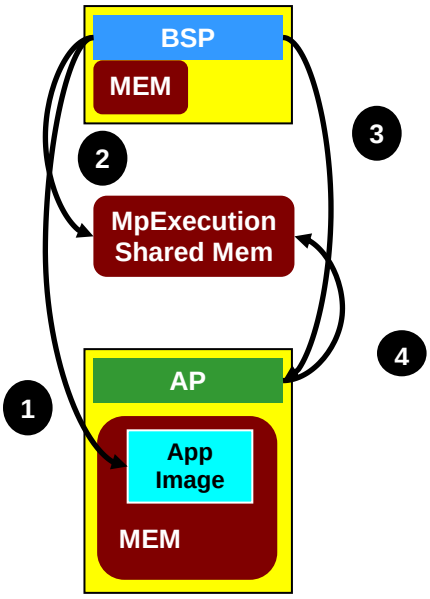
Memory partitioning currently is done in FlatPages, non-paging build only.

Currently, the memory is distributed for all processors in a way that a processor gets a chunk of memory. In the future, the memory should not be divided based on the number of processors but based on the number of kernels as a kernel could have one or more processors sharing the same chunk of memory.

Another topic of discussion is how we should have pure exclusive AP address space. That is, even in the midst of failure (e.g. memory corruption), an AP kernel or application should not able to access other AP kernel's memory. The Hive system actually protects that by having some kind of hardware support, which they call it "firewall" [Chapin'95]. Another way is probably to have one page-table for each AP kernel to monitor memory accesses and prevent memory access violation.

4.2. Running Applications on AP

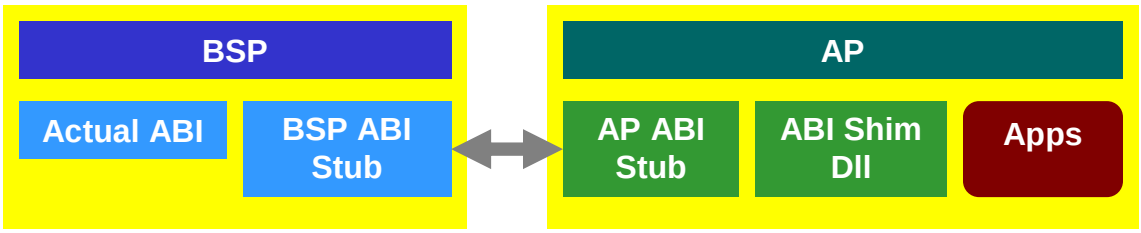
The second support is having the ability run applications on AP processors. The figure below illustrates how currently we run applications on AP. First, the BSP kernel has the ability to load an application image in the address space dedicated to another CPU. Next, the BSP kernel stores the entry point in the shared memory MpExecution context. In the future, the shared memory MpExecution context will be changed with hardware support on message passing. Third, BSP kernel sends IPI to the AP kernel on the corresponding AP processor. Finally, the AP kernel picks up the entry point from MpExecution context and starts executing at the specified entry point.



4.3. ABI Stubs

Since the AP kernel does not have the full OS services, some ABI calls must be rerouted to the BSP kernel. An ABI shim has been developed to allow applications running on non-bootstrap processors to make ABI calls. The AP kernel will have the ABI shim that sends messages via inter-processor interrupt. This is done by accomplishing several stuffs: First, we have the ABI shim implemented as DLL that gets dynamically linked to by application running on AP. The ABI shim will call AP ABI stub functions implemented in the AP kernel. These stub functions at the AP side are responsible to marshal the ABI arguments and send IPI to BSP kernel. The BSP ABI stub receives the ABI call interrupt, and executes the actual ABI. After the actual call completes, the BSP stub marshal back the return value and arguments for AP stub and send IPI to the corresponding AP.

The graph and code below illustrate the process described above.



Application: First, we have an application (TestMpAbi.x86) that makes an ABI call AbiX().

```
// In TestMpAbi.x86
int main () {
    return ProcessService::AbiX(args);
}
```


ABI Shim Dll: Next we have an ABI Shim dll (MpSyscalls.x86) that has the ABI implementation, and in this case, the implementation of AbiX(). The implementation contains a call to the AP ABI stub implemented in the AP kernel. In this case, the AP stub is StubAbiX().

```
// In MpSyscalls.x86
__declspec(dllexport) int __fastcall ProcessService::AbiX(args)
{
    return
    ProcessService::StubAbiX(args);
}
```

Originally, all imports in the application are resolved with the ABI exports from the full kernel (kernel.x86). However, since the application (TestMpAbi) is to run on non-bootstrap processors, the Singularity loader will resolved all imported ABI calls with the exported ABI calls from the ABI dll (MpSyscalls.x86).

The stub imports in the ABI shim (MpSyscalls.x86) are resolved with the stub exports from the AP kernel during PEImage initialization. Note that currently there is no such thing as AP kernel; All we have is BSP kernel.

AP ABI stub: The AP stub shown below is responsible for marshaling the arguments and send an ABI call IPI to the BSP. After that, at this point, it waits. If AP scheduler comes into play, this will actually yield to allow other threads to continue running.

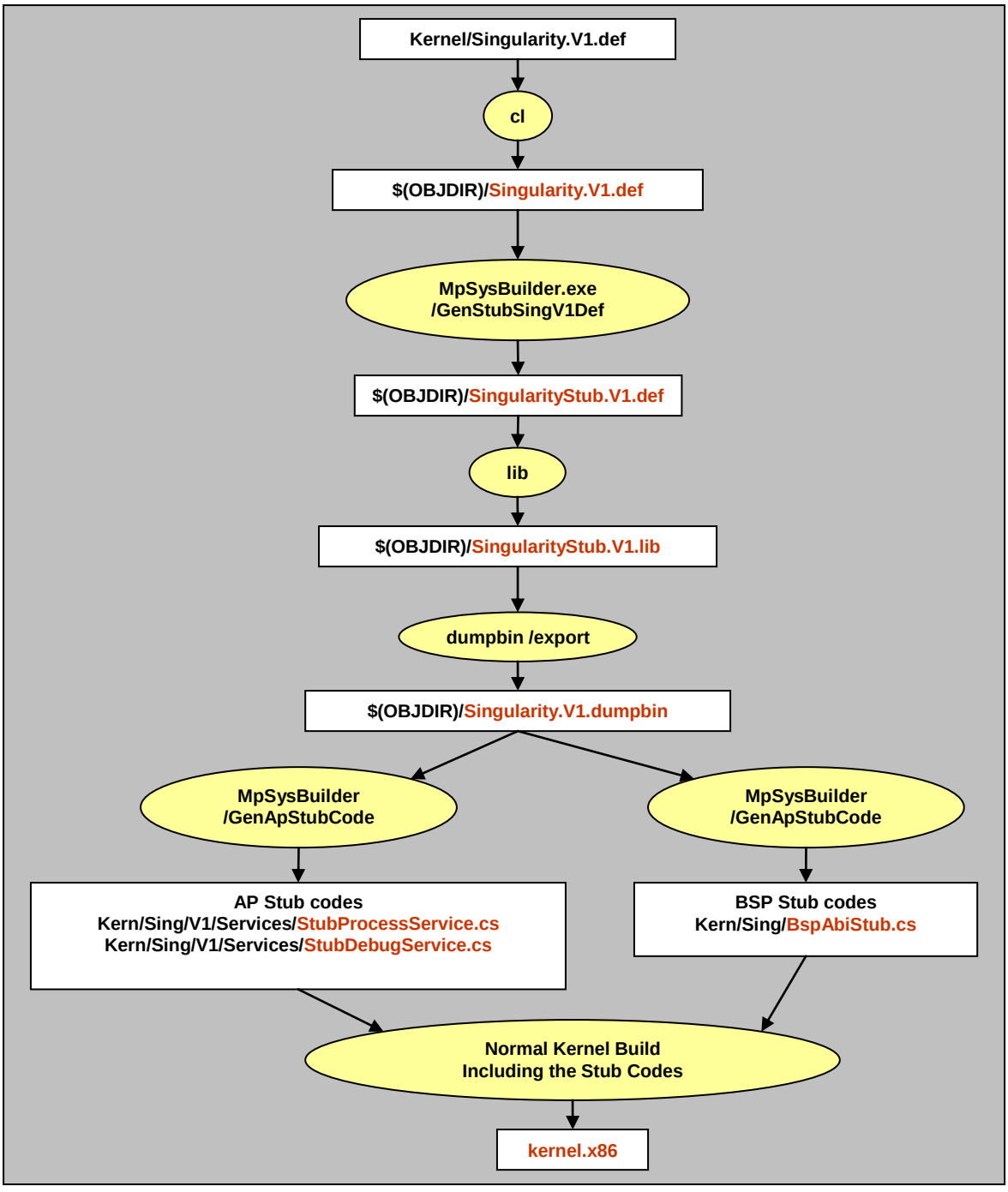
```
// In ApKernel.x86
int ProcessService::StubAbiX(args)
{
    AbiCall abiCall = ReserveAbiCall();
    abiCall.args = args;
    abiCall.abiNumber = STUB_ABI_X_NUMBER;
    SendAbiCall(); // IPI
    wait();
    retval = abiCall.retval;
    args = abiCall.args;
    return retval;
}
```

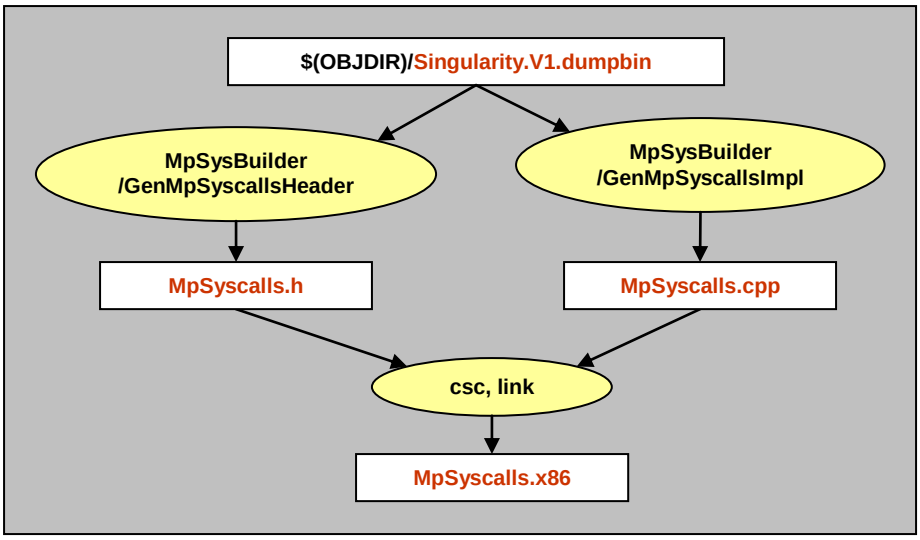
BSP ABI stub: The BSP kernel receives the AbiCall IPI and calls the corresponding BSP ABI stub (e.g. BspAbiStubX()). The BSP ABI stub is responsible in taking the ABI arguments, calling the actual ABI call and eventually marshalling the return value and arguments back to the corresponding AP.


```
// In BspKernel.x86 (kernel.x86 currently)
Processor::DispatchInterrupt()
{
    if (interrupt == Vector.AbiCall) {
        abiCall = GetAbiCall();
        BspAbiStub::ProcessAbiCall(abiCall);
    }
}

BspAbiStub::ProcessAbiCall(abiCall)
{
    Switch (abiCall.abiNumber) {
        Case STUB_ABI_X_NUMBER: BspAbiStubX(abiCall)

    }
}

BspAbiStub::BspAbiStubX(abiCall)
{
    args = abiCall.args;
    retval = AbiX(args);
    abiCall.retval = retval;
}

ProcessService::AbiX(args)
{
  // actual abi implementation
}
```

MpSyscallBuilder: The code for ABI shim dll, AP stubs, and BSP stubs, shown above are mostly automatically generated by MpSyscallBuilder.exe.

Below is the graph showing the process in generating the stubs.





5. References

[Balakrishnan'05] *Saisanthosh Balakrishnan (UWisc), Ravi Rajwar, Mike Upton, Konrad Lai (Intel)*, **The Impact of Performance Asymmetry in Emerging Multicore Architectures**, ISCA'05

[Chapin'95] *John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta (Stanford)*, **Hive: Fault Containment for Shared-Memory Multiprocessors**, SOSP'95

[Kumar'03] *Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, Keith I. Farkas*, **Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction**, MICRO'03

[Muir'98] *Steve Muir, Jonathan Smith (UPenn)*, **AsyMOS - An Asymetric Multi-processor Operating System**, OpenArch'98

[McAuley'03] *Derek McAuley, Rolf Neugebauer (Intel Cambridge)*, **A case for Virtual Channel Processors**, Position paper in SIGCOMM'03

